ASYNC JS, PROMISES, & REQUESTS

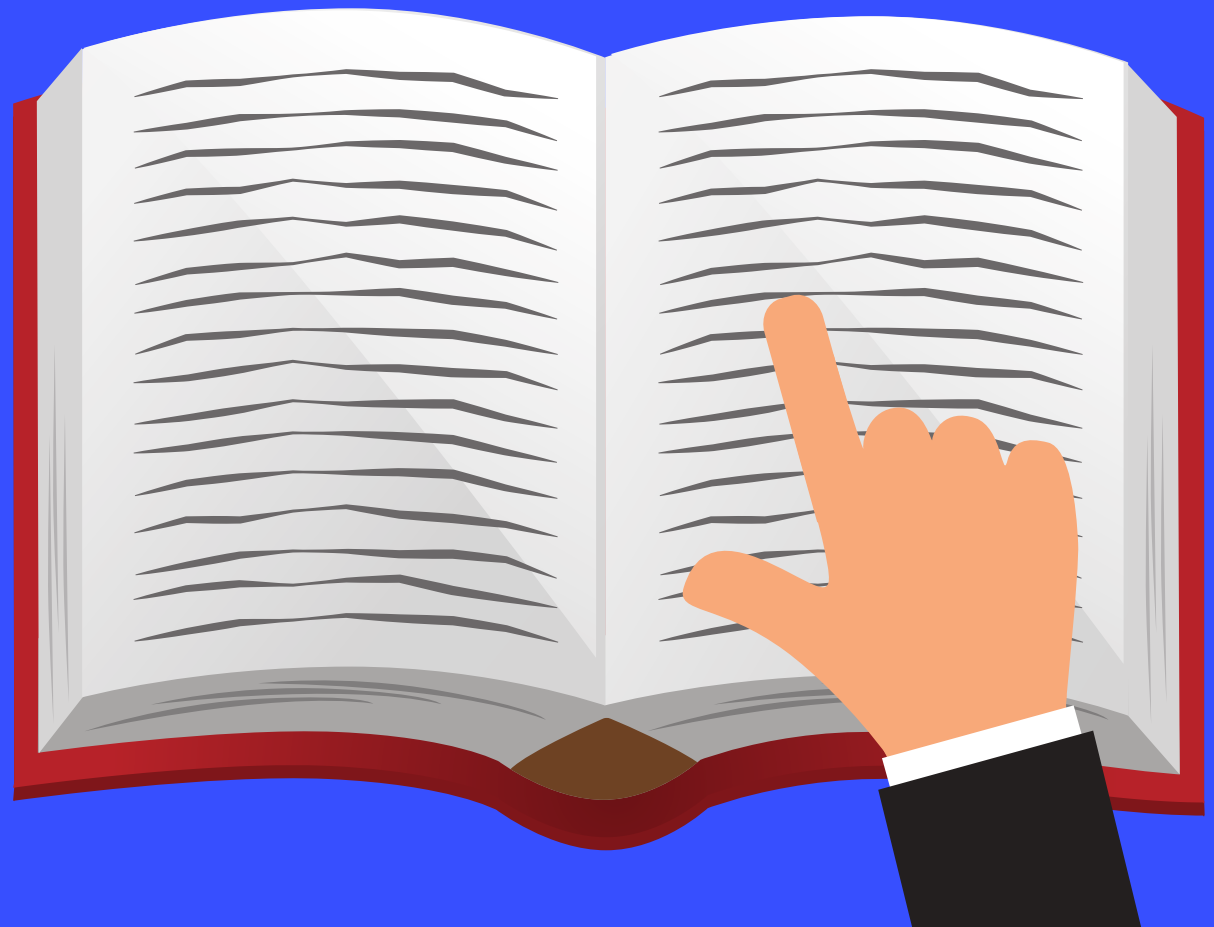
THE CALL STACK

CALL STACK

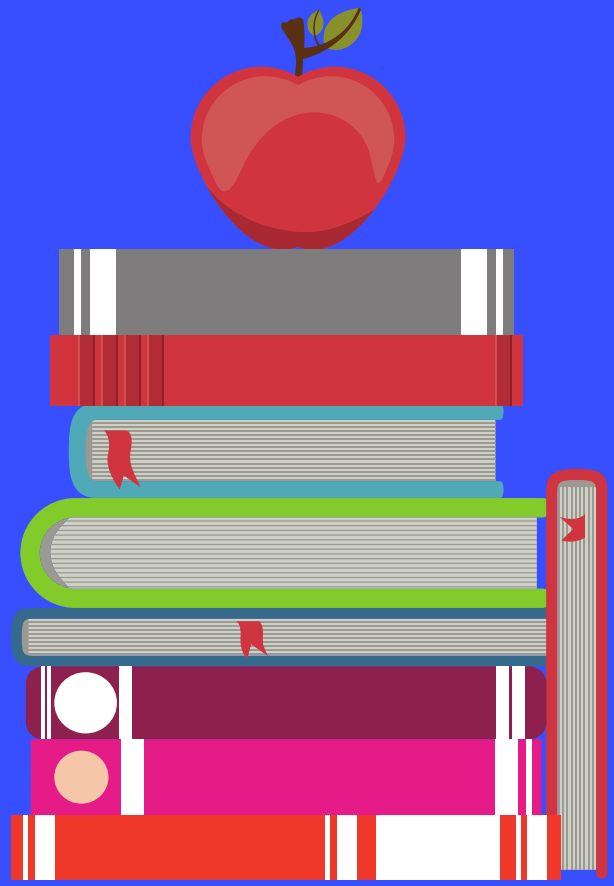
The mechanism the JS interpreter uses to keep track of its place in a script that calls multiple functions.

How JS "knows" what function is currently being run and what functions are called from within that function, etc.

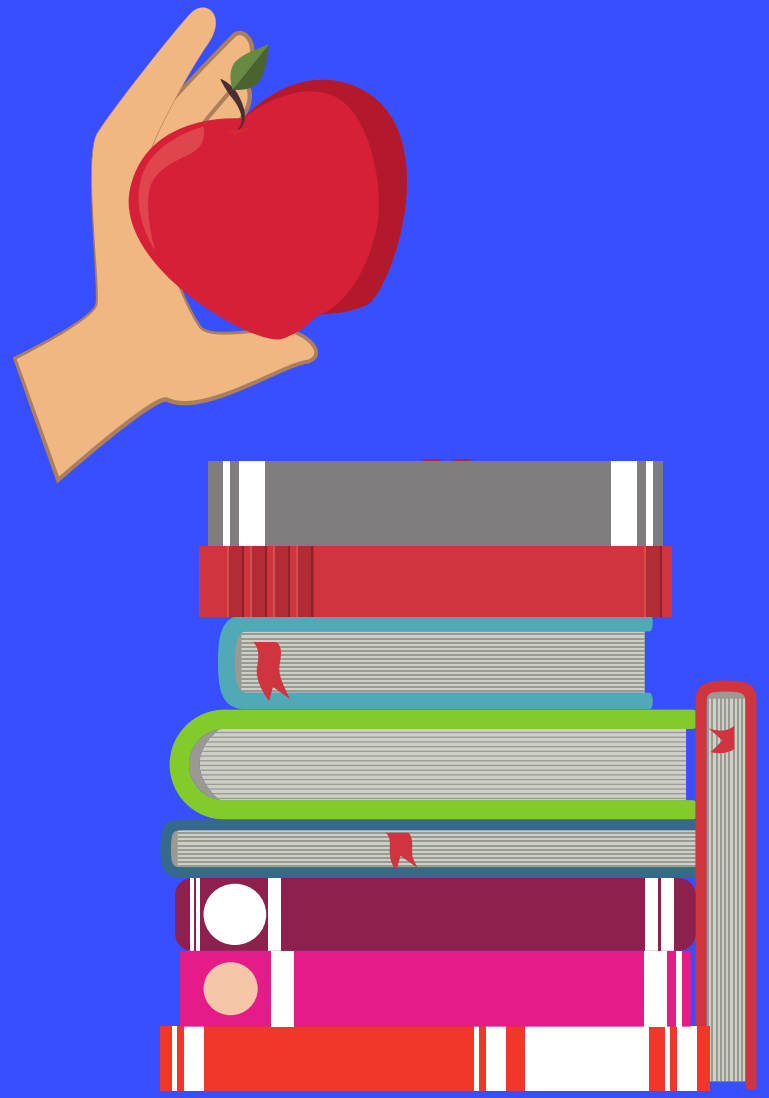
CALL STACK



LAST THING IN...



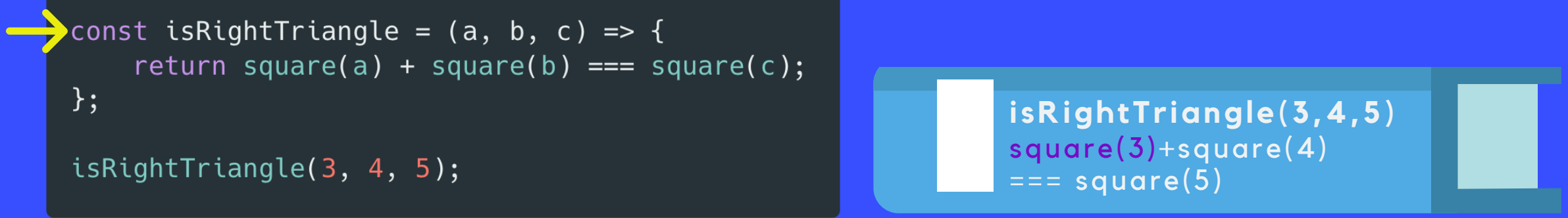
FIRST THING OUT...



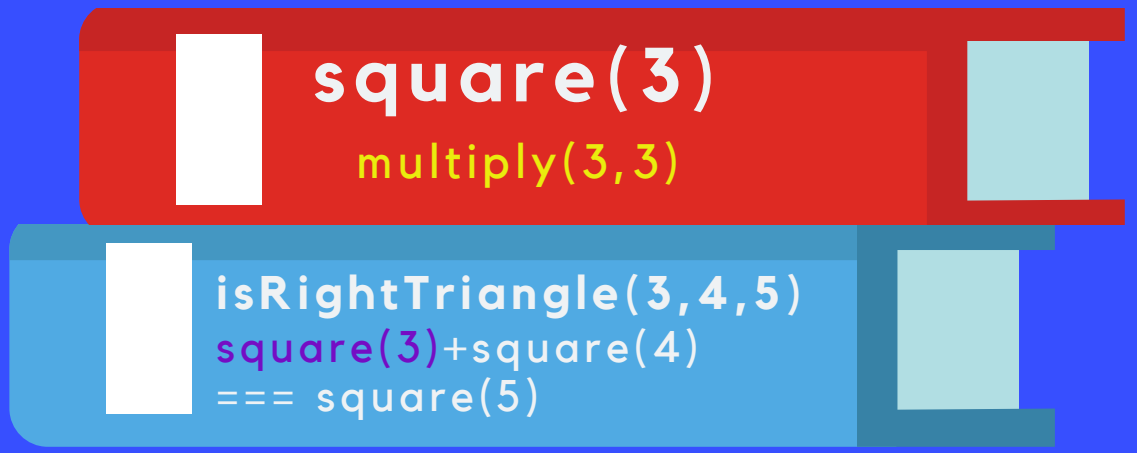
HOW IT WORKS

- When a script calls a function, the interpreter adds it to the call stack and then starts carrying out the function.
- Any functions that are called by that function are added to the call stack further up, and run where their calls are reached.
- When the current function is finished, the interpreter takes it off the stack and resumes execution where it left off in the last code listing.

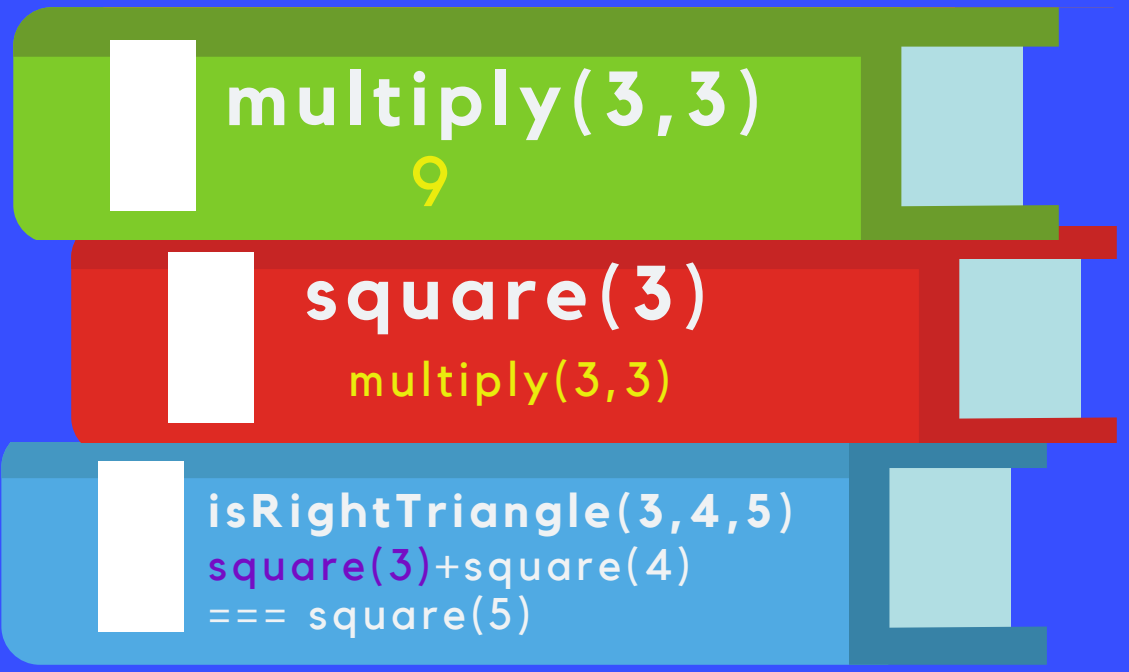
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



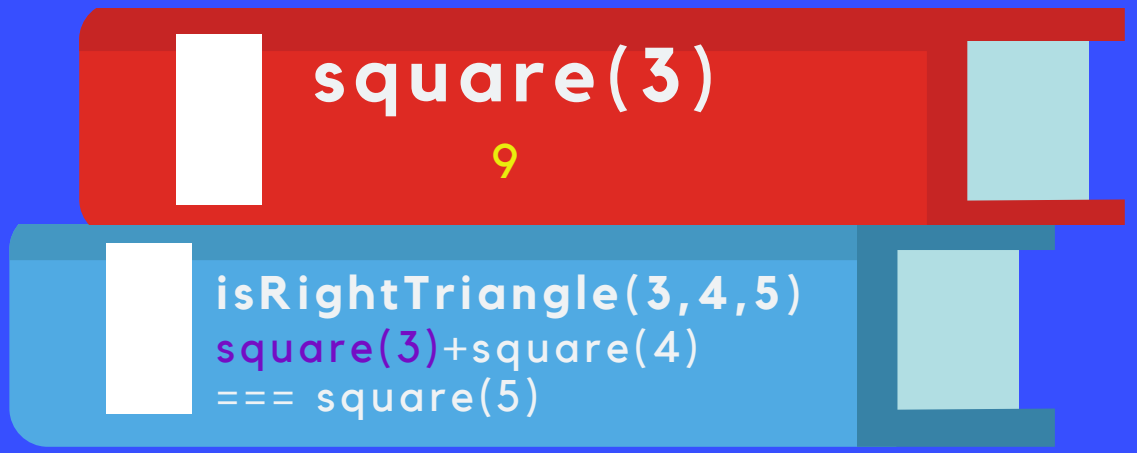
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



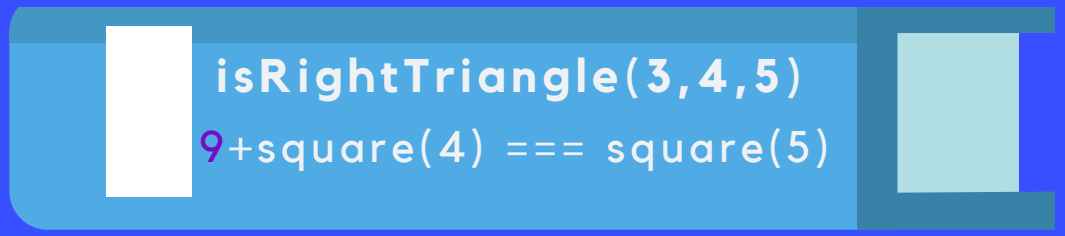
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



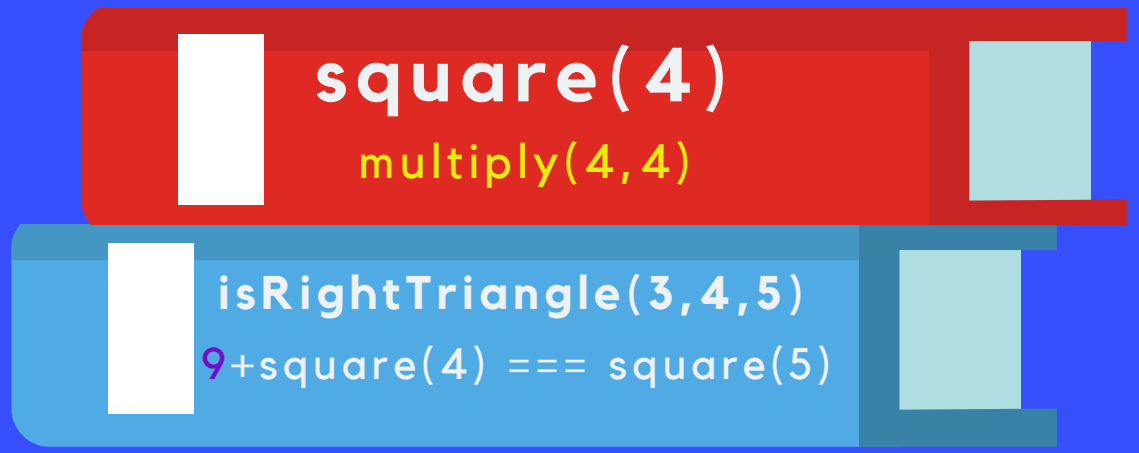
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



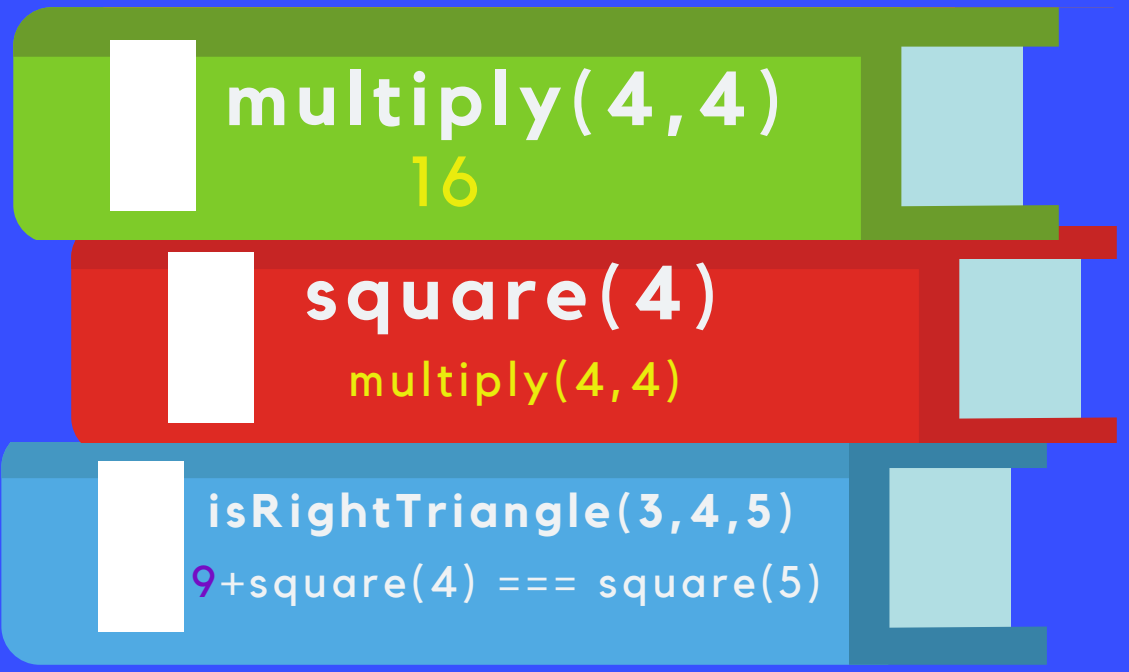
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



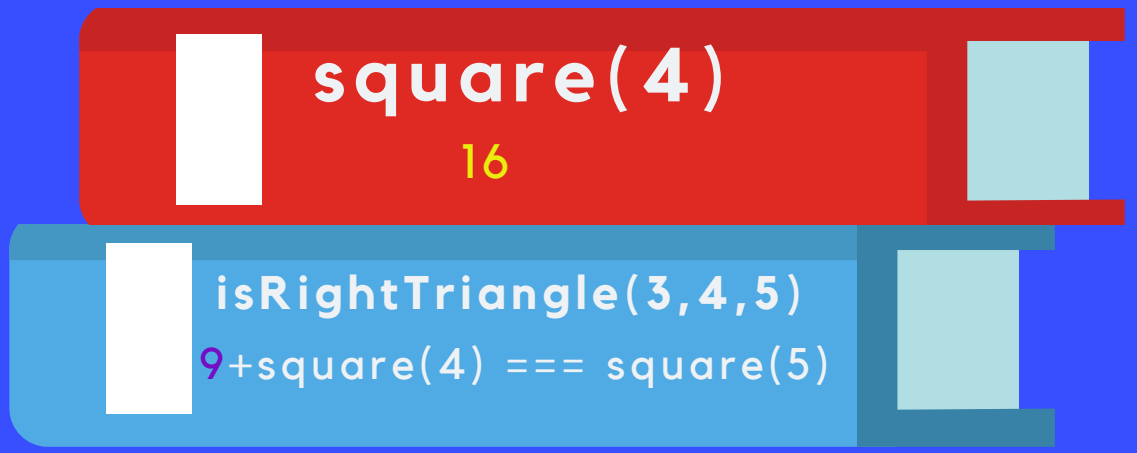
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



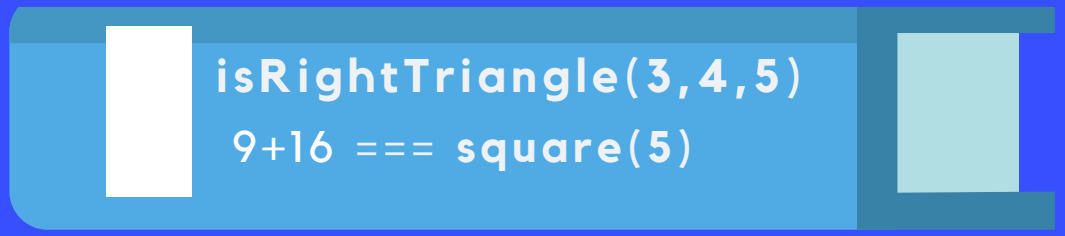
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



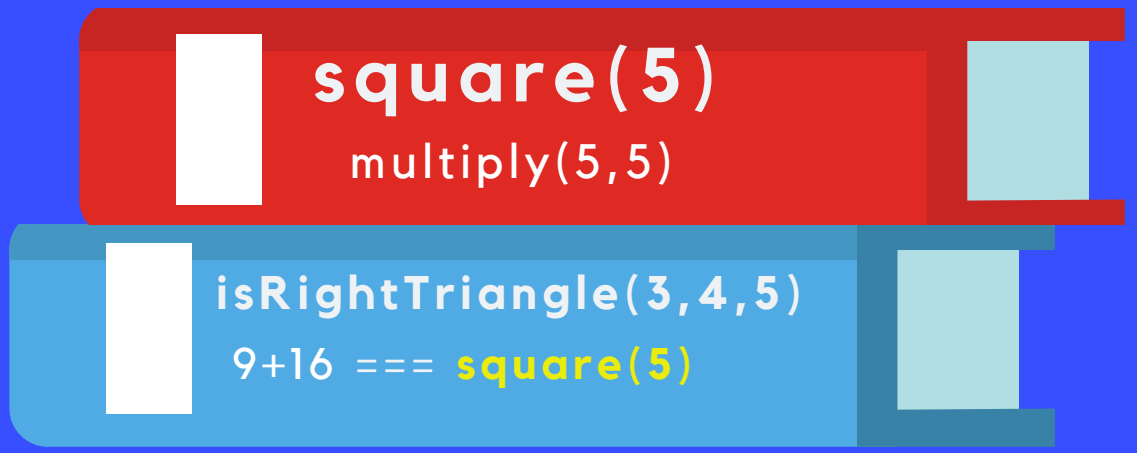
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



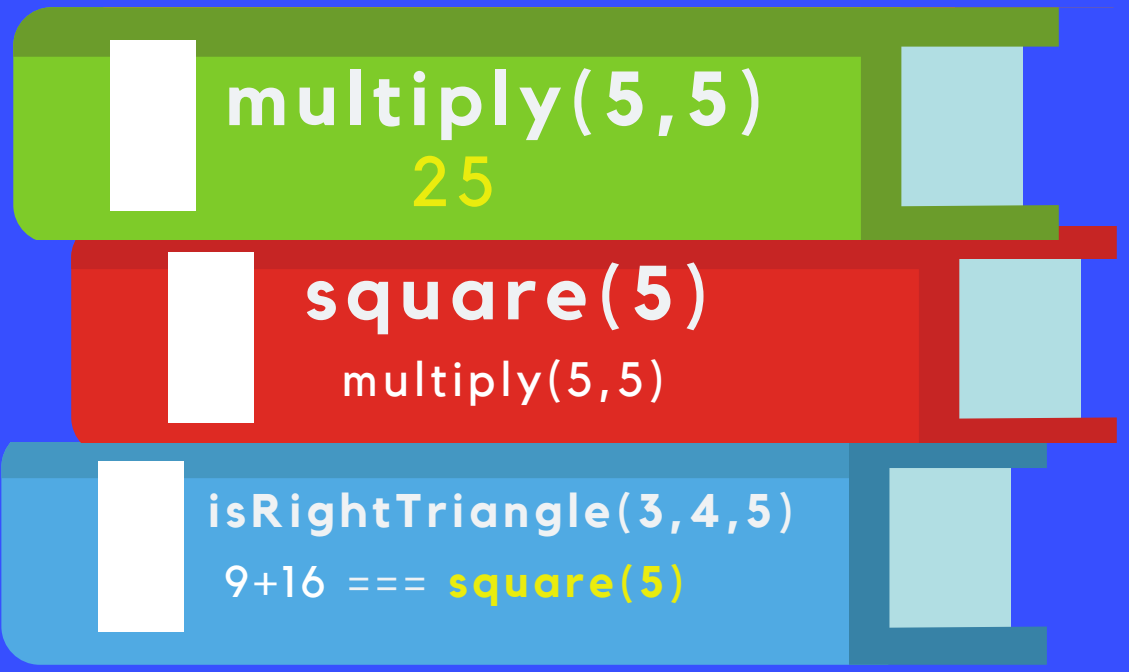
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



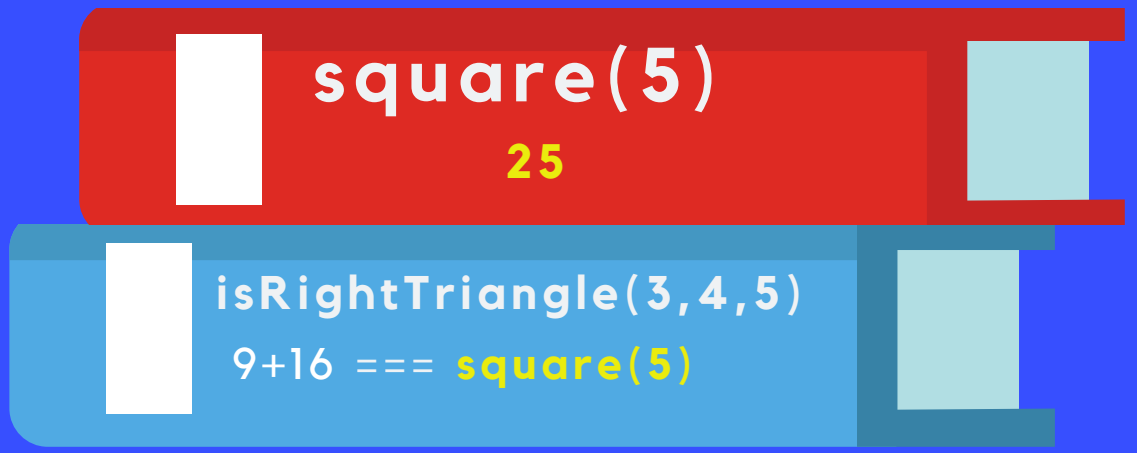
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



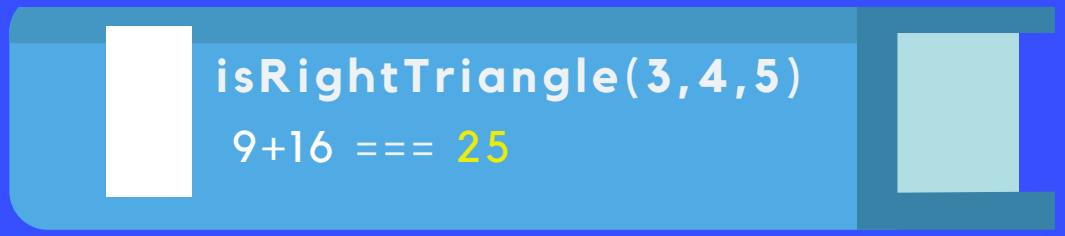
```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```



```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```

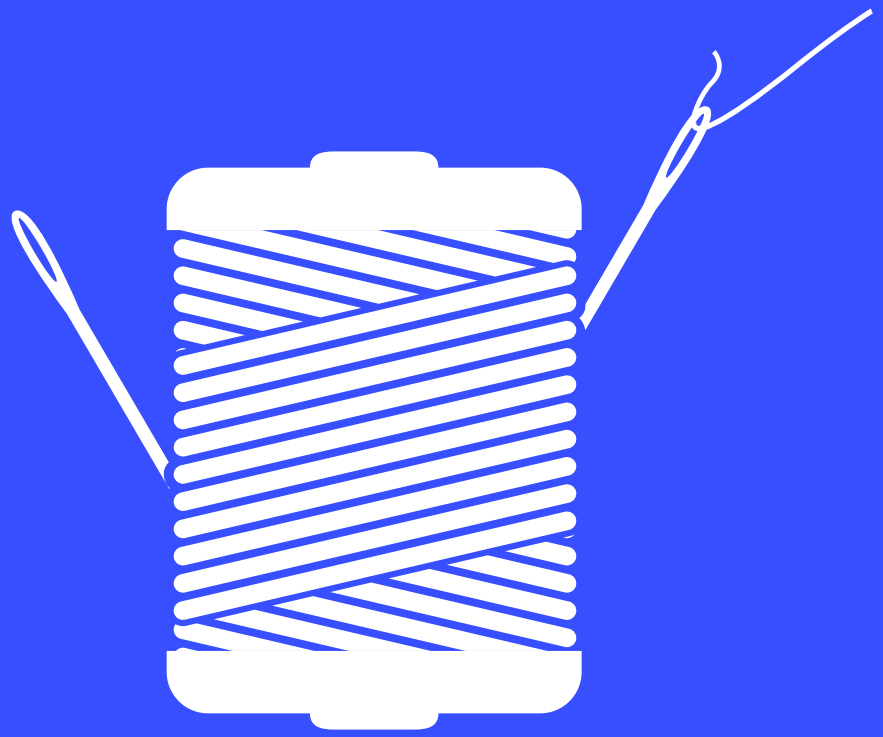


```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```

```
const multiply = (x, y) => x * y;

const square = (x) => multiply(x, x);

const isRightTriangle = (a, b, c) => {
    return square(a) + square(b) === square(c);
};

isRightTriangle(3, 4, 5);
```

true

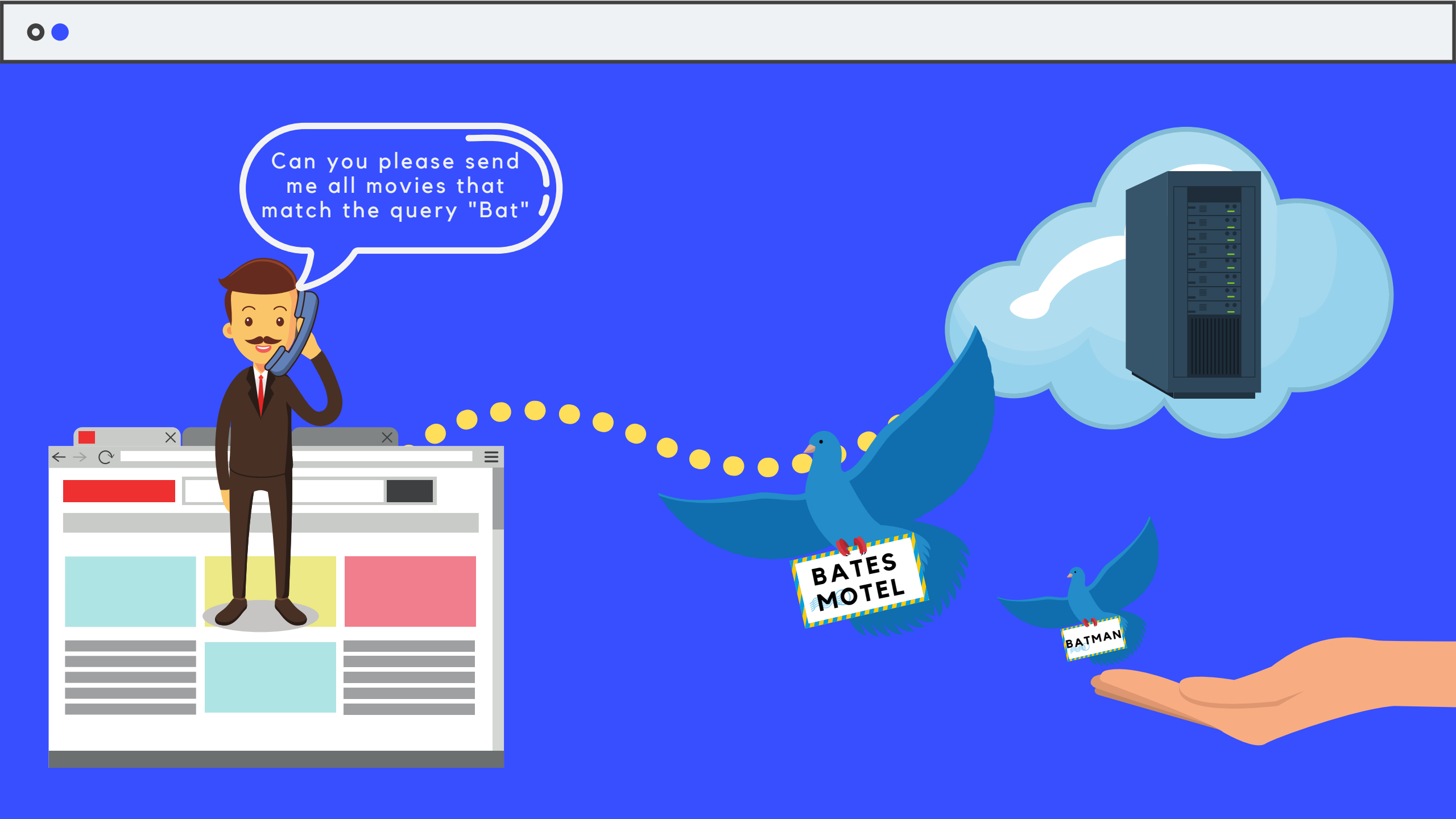
JS IS SINGLE THREADED



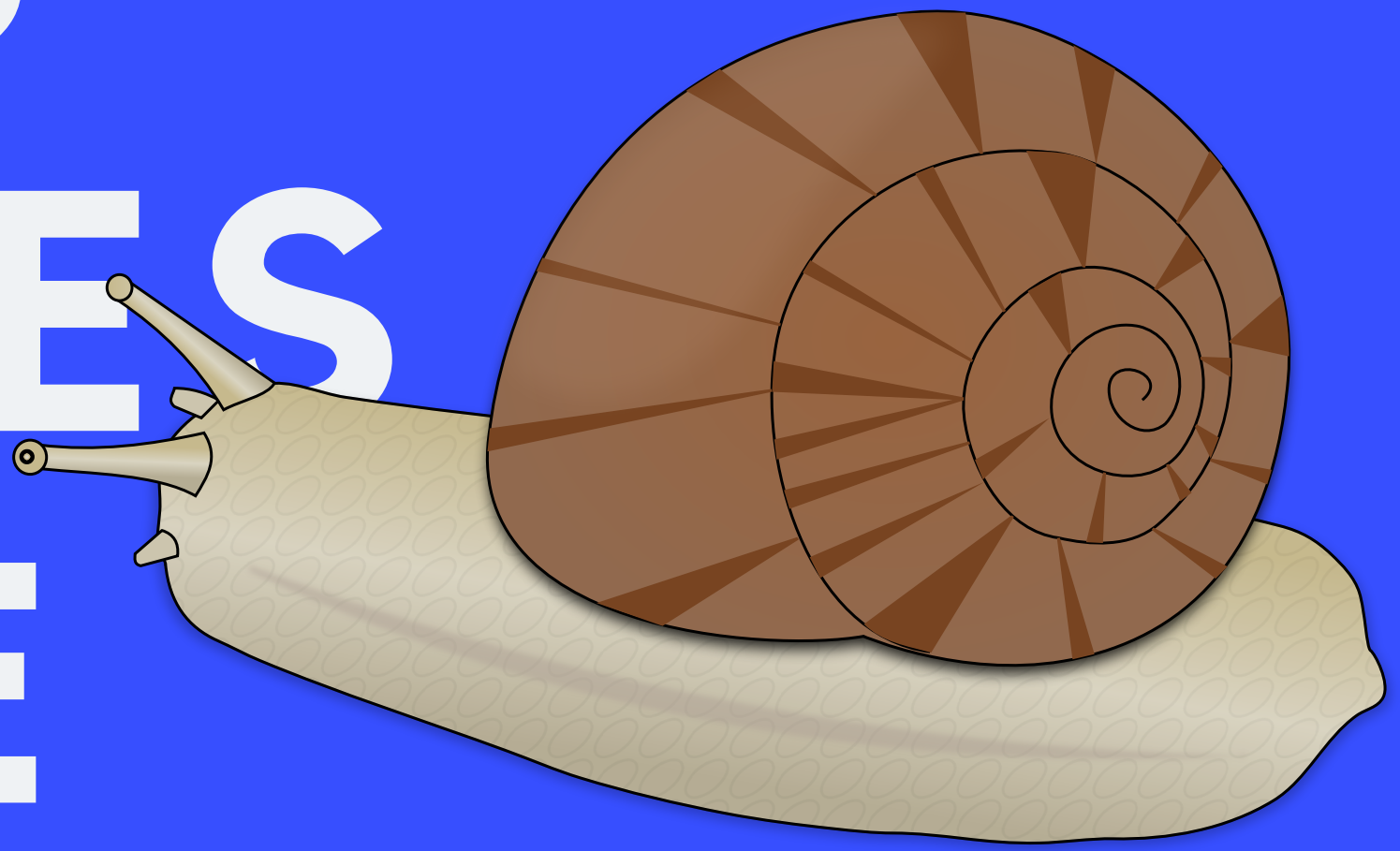
WHAT DOES THAT MEAN?

At any given point in time, that single JS thread is running at most one line of JS code.

Can you please send me all movies that match the query "Bat"


Can you please send me all movies that match the query "Bat"
BATES MOTEL
BATMAN

THIS TAKES TIME



IS OUR APP GOING TO GRIND TO A HALT?

What happens when something takes a long time?

```
const newTodo = input.value; //get user input
saveToDatabase(newTodo); //this could take a while!
input.value = '';  //reset form
```

Fortunately...
We have a workaround

```
console.log('I print first!');
setTimeout(() => {
    console.log('I print after 3 seconds');
}, 3000);
console.log('I print second!');
```

CALLBACKS???!

HOW DOES IT WORK ?

THE BROWSER DOES THE WORK!

TO THE RESCUE

OK BUT HOW?

- Browsers come with Web APIs that are able to handle certain tasks in the background (like making requests or setTimeout)
- The JS call stack recognizes these Web API functions and passes them off to the browser to take care of
- Once the browser finishes those tasks, they return and are pushed onto the stack as a callback.

A CLOSER LOOK

> I print first!

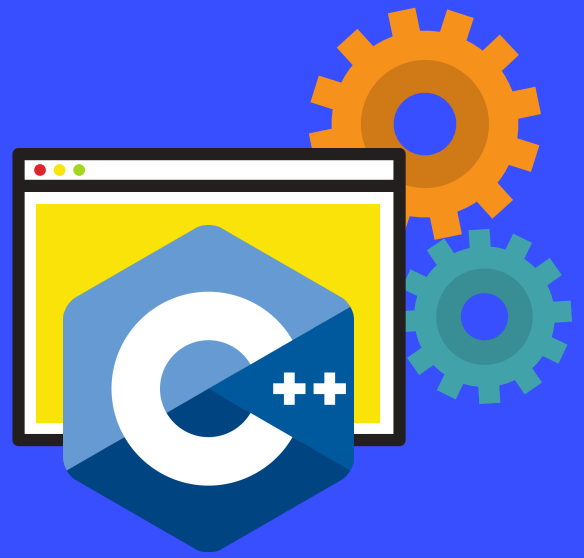
```
console.log('I print first!');
setTimeout(() => {
    console.log('I print after 3 seconds');
}, 3000);
console.log('I print second!');
```

```
> I print first!
```

Hey browser, can you set a timer for 3 seconds?

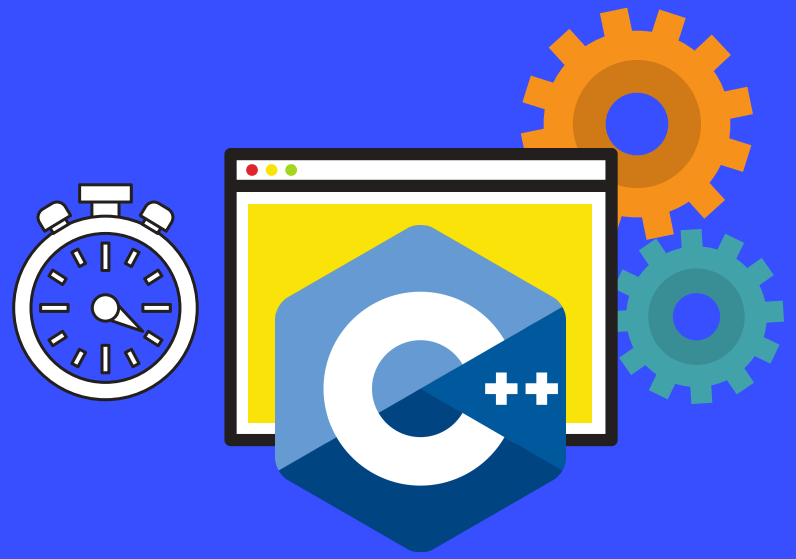
```
console.log('I print first!');
setTimeout(() => {
    console.log('I print after 3 seconds');
}, 3000);
console.log('I print second!');
```

OKEEEDOKEEE



```
> I print first!
> I print second!
```

```
console.log('I print first!');
setTimeout(() => {
    console.log('I print after 3 seconds');
}, 3000);
console.log('I print second!');
```

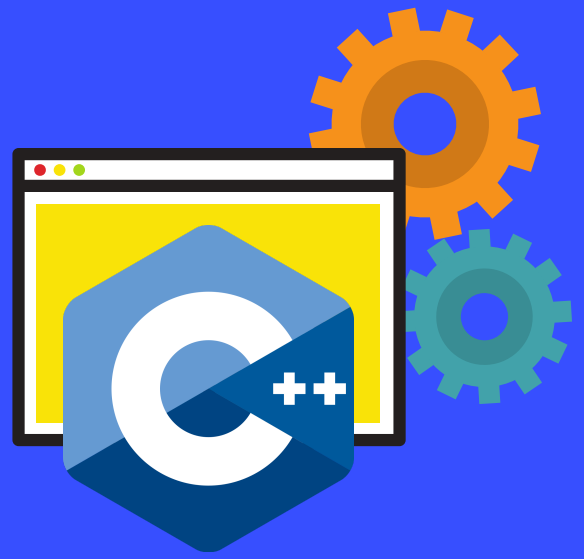


```
> I print first!
> I print second!
```

```
console.log('I print first!');
setTimeout(() => {
    console.log('I print after 3 seconds');
}, 3000);
console.log('I print second!');
```

```
> I print first!
> I print second!
> I print after 3 seconds!
```

```
console.log('I print first!');
setTimeout(() => {
    console.log('I print after 3 seconds');
}, 3000);
console.log('I print second!');
```



```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename,
function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

Callback Hell

ENTER PROMISES

A Promise is an object representing the eventual completion or failure of an asynchronous operation

PROMISES

A pattern for writing async code.

RESOLVE

REJECT

A promise is a returned object to which you attach callbacks, instead of passing callbacks into a function

loadRedditPosts (not shown) returns a promise

```
loadRedditPosts('/r/funny')
    //this runs if promise is resolved:
    .then((res) => {
        console.log(res.data);
  })
  //this runs if promise is rejected:
    .catch((err) => {
        console.log('Oh No!', err);
    });
```

This function returns a Promise which is randomly resolved/rejected.

```
const makeFakeRequest = () => {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const randNum = Math.random();
            if (randNum > 0.5) resolve({ data: 'lol', status: 200 });
            reject({ status: 404, data: 'NO DICE' });
        }, 1000);
    });
};
```